LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

MIT/LCS/TM-410

ONLINE TRACKING OF MOBILE USERS

Baruch Awerbuch
David Peleg

August 1989

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

Online tracking of mobile users

Baruch Awerbuch * David Peleg †

Abstract

This paper deals with the problem of maintaining a distributed directory server, that enables us to keep track of mobile users in a distributed network. The paper introduces the graph-theoretic concept of *regional matching*, and demonstrates how finding a regional matching with certain parameters enables efficient tracking. A polynomial-time algorithm that constructs such a regional matching is presented. The communication overhead of our tracking mechanism is within a polylogarithmic factor of the lower bound.

Key words:

Communication networks, mobile users, communication complexity, amortized analysis.

*Dept. of Mathematics and Lab. for Computer Science, M.I.T., Cambridge, MA 02139; ARPANET: baruch@theory.lcs.mit.edu. Supported by Air Force Contract TNDGAFOSR-86-0078, ARO contract DAAL03-86-K-0171, NSF contract CCR8611442, and a special grant from IBM.

†Department of Applied Mathematics, The Weizmann Institute, Rehovot 76100, Israel. Supported in part by an Alon Fellowship.

1 Introduction

This paper deals with the problem of tracking mobile users in a distributed communication network. When users are allowed to move from one network vertex to another, it is necessary to have a mechanism enabling one to keep track of such users and contact them at their current residence. The purpose of this work is to design efficient tracking mechanisms, based on distributed directory structures (cf. [LEH]), minimizing the communication redundancy involved.

Networks with mobile users are by no means far-fetched. A prime example is that of cellular phone networks. In fact, one may expect that in the future, all telephone systems will be based on "mobile telephones numbers," i.e., ones that are not bound to any specific physical location. Another possible application is a system one may call "distributed yellow pages," or "distributed match-making" [MV, KV]. Such a system is necessary in an environment consisting of mobile "servers" and "clients." The system has to provide means for enabling clients in need of some service to locate the whereabouts of the server they are looking for. (Our results are easier to present assuming the servers are distinct. However, they are applicable also to the case when a user is actually looking for one of the closest among the set of identical servers.)

In essence, the tracking mechanism has to support two operations: a "move" operation, causing a user to move to a new destination, and a "find" operation, enabling one to contact the current address of a specified user. However, the tasks of minimizing the communication overhead of the "move" and "find" operations appear to be contradictory to each other. This can be realized by examining the following two extreme strategies (considered also in [MV]).

The *full-information* strategy requires every vertex in the network to maintain a complete directory containing up-to-date information on the whereabouts of every user. This makes the "find" operations cheap. On the other hand, "move" operations are very expensive, since it is necessary to update the directories of all vertices. Thus this strategy is appropriate only for a near static setting, where users move relatively rarely, but frequently converse with each other.

In contrast, the *no-information* strategy opts not to perform any updates following a "move," thus abolishing altogether the concept of directories and making the "move" operations cheap. However, establishing a connection via a "find" operation becomes very expensive, as it requires a global search over the entire network. Alternatively, trying to to eliminate this search, it is possible to require that whenever a user moves, it leaves a "forwarding" pointer at the old address, pointing to the new address. Unfortunately, this

heuristic still does not guarantee any good worst-case bound for the “find” operations.

Our purpose is to design some intermediate “partial-information” strategy, that will perform well for any communication/travel pattern, making the costs of both “move” and “find” operations relatively cheap. This problem was tackled also by [MV, KV]. However, their approach was to consider only the global worst-case performance. Consequently, the schemes designed there treat all requests alike, and ignore considerations such as locality. Our goal is to design more refined strategies that take into account the inherent costs of the particular requests at hand. It is clear that in many cases these costs may be lower than implied by the global worst-case analysis. In particular, we would like moves to a near-by location, or searches for near-by users, to cost less. (Indeed, consider the case of a person who moves to a different room in the same hotel. Clearly, it is wasteful to update the telephone directories from coast to coast, and notifying the hotel operator should normally suffice.) Thus we are interested in the worst case *overhead* incurred by a particular strategy. This overhead is evaluated by comparing the total cost invested in a sequence of “move” and “find” operations against the inherent cost (namely, the cost incurred by the operations themselves, assuming full information is available for free.) This comparison was done over all sequences of “move” and “find” operations. The strategy proposed in this paper guarantees overheads that are *polylogarithmic* in the size of the network.

The rest of the paper is organized as follows. The next section contains a precise definition of the model and the problem. In Section 3 we give an overview of the proposed solution. The directory mechanism is described in Section 4. Its complexity and correctness are analyzed in Section 5. This mechanism is based on the availability of a structure called a *regional matching* in the graph. The construction of a regional matching is described in Section 6 and analyzed in Section 7. Finally, Section 8 concludes with a discussion.

2 The problem

2.1 The model

We consider the standard model of a point-to-point communication network. The network is described by a connected undirected graph $G = (V, E)$, $|V| = n$. The vertices of the graph represent the processors of the network and the edges represent bidirectional communication channels between the vertices. A vertex may communicate directly only with its neighbors, and messages to non-neighboring vertices u and v are sent along some path connecting them in the graph.

We assume the existence of a *weight* function $w : E \rightarrow \mathcal{R}$, assigning an arbitrary non-negative weight $w(e)$ to each edge $e \in E$. For two vertices u, w in G, let $dist_G(u, w)$ denote the length of a shortest path in G between those vertices, where the length of a path $(e_1, \ldots, e_s)$ is $\sum_{1 \leq i \leq s} w(e_i)$. For two sets of vertices U, W in G, let $dist_G(U, W) = \min\{dist_G(u, w) \mid u \in U, w \in W\}$. (We sometimes substitute w for a singleton $W = \{w\}$.) Let $Diam(G)$ denote the (weighted) *diameter* of the network G, namely, the maximal distance between any two vertices in G.

Communication complexity is measured as follows. The basic message length is $O(\log n)$ bits. Longer messages are charged proportionally to their length (i.e., a message of length $\ell > \log n$ is viewed as $\lceil \frac{\ell}{\log n} \rceil$ basic messages). The *communication cost* of transmitting a basic message over an edge e is the weight $w(e)$ of that edge. The communication cost of a *protocol* π, denoted $Cost(\pi)$, is the sum of the communication costs of all message transmissions performed during the execution of the protocol. For simplicity, we assume that $Diam(G) = poly(n)$. Thus the distance between two vertices can be transmitted in a single message, as $\log Diam(G) = O(\log n)$.

We assume the existence of efficient routing facilities in the system. More specifically, assume that whenever a processor v wishes to send a message to a processor u, the message will be sent along a route as efficient as possible in the network, and the cost of the routing is $O(dist(u, v))$.

2.2 Statement of the problem

Denote by $Addr(\xi)$ the current address of a specific user ξ. A directory $\mathcal{D}$ with respect to the user ξ is a distributed data structure which enables to move and to find the user, namely, to perform the following two operations.

`Find`$(\mathcal{D}, \xi, v)$: invoked at the vertex v, this operation delivers a search message from v to the location $s = Addr(\xi)$ of the user ξ.

`Move`$(\mathcal{D}, \xi, s, t)$: invoked at the current location $s = Addr(\xi)$ of the user ξ, this operation moves ξ to a new location t and performs the necessary updates in the directory.

We assume that individual activations of the operations `Find` and `Move` do not interleave in time, i.e., are performed in an "atomic" fashion, and thus avoid the issues of concurrency control, namely, questions regarding the simultaneous execution of multiple `Find` / `Move` operations. These issues are briefly discussed in the concluding section.

2.3 Complexity measures

We are interested in measuring the communication complexity of the `Find` and `Move` operations in our directories. More specifically, we study the overheads incurred by our algorithms, compared to the minimal "inherent" costs associated with each `Find` and `Move` operation. Consequently, let us first identify these optimal costs.

Consider a `Move` instruction $M = \mathtt{Move}(\mathcal{D}, \xi, s, t)$. Let $Reloc(\xi, s, t)$ denote the actual relocation cost of the user ξ from s to t. We define the *optimal cost* of the operation M as $Opt_cost(M) = Reloc(\xi, s, t)$, which is the inherent cost assuming no extra operations, such as directory updates, are taken. This cost depends on the distance between the old and new location, and we assume it satisfies $Reloc(\xi, s, t) \geq dist(s, t)$. (In fact, the relocation of a server is typically much more expensive than just sending a single message between the two locations.)

Now consider a `Find` instruction $F = \mathtt{Find}(\mathcal{D}, \xi, v)$. Define the optimal cost of F as $Opt_cost(F) = dist(v, Addr(\xi))$.

Recall that $Cost(F)$ (respectively, $Cost(M)$) denotes the actual communication cost of the operation F (resp., M).

We would like to define the "amortized overhead" of our operations, compared to their optimal cost. For that purpose we consider mixed sequences of `Move` and `Find` operations. Given such a sequence $\bar{\sigma} = \sigma_1, \ldots, \sigma_\ell$, let $\mathcal{F}(\bar{\sigma})$ denote the subsequence obtained by picking only the `Find` operations from $\bar{\sigma}$, and similarly let $\mathcal{M}(\bar{\sigma})$ denote the subsequence obtained by picking only the `Move` operations from $\bar{\sigma}$ (i.e., $\bar{\sigma}$ consists of some shuffle of these two subsequences).

Define the optimal cost and the cost of the subsequence $\mathcal{F}(\bar{\sigma}) = (F_1, \ldots, F_k)$ in the natural way, setting

$$\begin{aligned} Opt_cost(\mathcal{F}(\bar{\sigma})) &= \sum_{i=1}^{k} Opt_cost(F_i) \\ Cost(\mathcal{F}(\bar{\sigma})) &= \sum_{i=1}^{k} Cost(F_i). \end{aligned}$$

The *find-stretch* of the directory with respect to a given sequence of operations $\bar{\sigma}$ is defined as

$$Stretch_{find}(\bar{\sigma}) = \frac{Cost(\mathcal{F}(\bar{\sigma}))}{Opt_cost(\mathcal{F}(\bar{\sigma}))}.$$

The find-stretch of the directory, denoted $Stretch_{find}$, is the least upper bound on $Stretch_{find}(\bar{\sigma})$, taken over all finite sequences $\bar{\sigma}$.

For the subsequence $\mathcal{M}(\bar{\sigma})$, define the optimal cost $Opt_cost(\mathcal{M}(\bar{\sigma}))$, the cost $Cost(\mathcal{M}(\bar{\sigma}))$ and the *move-stretch* factors $Stretch_{move}(\bar{\sigma})$ and $Stretch_{move}$ analogously.

We comment that our definitions ignore the initial set-up costs involved in organizing the directory when the user first enters the system.

Finally, define the memory requirement of a directory as the total amount of memory bits it uses in the processors of the network.

2.4 Main results

Our main result is the construction of a dynamic directory server, $\mathcal{D}$, guaranteeing $Stretch_{find} = O(\log^2 n)$ and $Stretch_{move} = O(\log^2 n)$, and requiring a total of $O(n \log^3 n + N \log^2 n)$ bits of memory (including both data and bookkeeping information) throughout the network, for handling N users.

3 Overview of the solution

Our scheme is based on a distributed data structure storing pointers to the locations of each user in various vertices. These pointers are updated as users move in the network. In order to localize the update operations on the pointers, we allow some of these pointers to be out of date. That is, we frequently update pointers at nearby locations but rarely update pointers at distant ones.

Formally, our dynamic directory $\mathcal{D}$ is composed of a hierarchy of $\delta = \lceil \log Diam(G) \rceil$ *regional directories* $\mathcal{RD}_i$, $1 \leq i \leq \delta$, covering successively larger portions of the network. The purpose of the regional directory $\mathcal{RD}_i$ at level i of the hierarchy is to enable a user to track any other user residing within distance 2^i from it. To prevent waste in performing a `Move` operation we use a mechanism of "forwarding addresses". Our update policy can be schematically described as follows. Whenever a user moves to a new location at distance d away, a pointer is left at the old location. Only the $\log d$ lowest levels of the hierarchy of the dynamic directory are updated, so that nearby searchers would be able to locate the right address directly. Searchers from distant locations fail in locating the user using the lower-level regional directories (since they are in a different region). They therefore have to use higher levels of the hierarchy. These levels will indeed have some information on the searched user, but this information may be out of date, and lead to some old location. The searcher will then be redirected to the new location through a chain of forwarding pointers. The

crucial point is that updates at the low levels are *local*, and thus require low communication complexity.

The 2^i-regional directory is implemented as follows. As in the match-making strategy of [MV], the process is based on intersecting "read" and "write" sets. A vertex v reports about every user it hosts to all vertices in some specified *write set*, $Write(v)$. While looking for a particular user, the searching vertex w queries all the vertices in some specified *read set*, $Read(w)$. We define the graph-theoretic concept of a 2^i-regional matching as a collection consisting of a read set and a write set for each vertex, with the property that the read set of a vertex intersects with the write set of any vertex within distance 2^i from it. (The match-making functions of [MV, KV] do not have any distance limitation, and they insist on having exactly one element in each intersection.) The relevant parameters of a regional matching are its *radius*, which the maximal distance from a vertex to any other vertex in its read or write set, and its *degree*, which is the maximal number of vertices in any read or write set. Now, the communication overhead of performing the `Find` and `Move` operations in a 2^i-regional directory grows as the product of the degree and the radius of the related 2^i-regional matching. There appears to be a trade-off between these two parameters, making simultaneous minimization of both of them a nontrivial task.

A related graph-theoretic problem is that of designing "sparse" graph covers, i.e., covering the graph by (not necessarily disjoint) clusters of vertices. The parameters of interest in that problem are the degrees of vertices in the cover (viewing it as a hypergraph on the set of vertices) and the maximal radius of cluster. Here, too, it is possible to trade-off radius for degree; for example, merging together a number of clusters, we may reduce the maximal degree at the expense of increasing the radius. Although the relationship between matchings and covers is not immediate (and in particular, the definitions of degree seem to measure different quantities), there is a strong connection between the two constructs, and given a cover it is possible to achieve a matching with the same degree and radius.

Algorithms for computing sparse covers are studied in [P1], and their applications to directory problems are proposed in [P2]. Unfortunately, the radius-degree trade-off achieved therein is not sufficient for the purposes of the current paper, since the maximal degree can only be bounded by $n^{c/\log r}$, where r is the maximal cluster radius and c is a constant. Trying to balance off the radius with the degree brings both parameters to the order of $2^{O(\sqrt{\log n})} = O(n^{O(\log\log n/\log n)})$.

However, it is much easier to construct a cover with small *average* degree. In the algorithm of [P1], the average degree drops more rapidly with the increase in the radius r, namely, as $n^{c/r}$. Thus, balancing off the average degree and the radius allows us to upper-bound both

parameters by $O(\log n)$.

In this paper, we show that an algorithm producing a cover with small *average* degree can be transformed into an algorithm producing a matching with small *maximal* degree, with the same radius. The idea is that in a cover with small average degree, "most" vertices have small degrees. For such vertices, we construct a "partial matching", and delete them from the graph. We repeat this algorithm for the remaining vertices, until no vertices are left. We show that this process has to be repeated only for a small ($k = \log n$) number of phases. In other words, we obtain a regional matching with small maximal degree from a *family* of covers with small average degrees.

4 The main construction

4.1 The concept of a regional matching

The basic components of our construction are a read set $Read(v) \subseteq V$ and a write set $Write(v) \subseteq V$, defined for every vertex v. Consider the collection $\mathcal{RW}$ of all pairs of sets, namely

$$\mathcal{RW} = \{ \, Read(v), \; Write(v) \mid v \in V \, \}.$$

Definition 4.1 The collection $\mathcal{RW}$ is an *m-regional matching* (for some integer $m \geq 1$) if for all $v, u \in V$ such that $dist(u, v) \leq m$, $Write(v) \bigcap Read(u) \neq \emptyset$.

For any m-regional matching $\mathcal{RW}$ define the following parameters.

$$\begin{aligned}
Deg_{write}(\mathcal{RW}) &= \max_{v \in V} |Write(v)| \\
Rad_{write}(\mathcal{RW}) &= \frac{1}{m} \cdot \max_{u,v \in V} \{dist(u,v) \mid u \in Write(v)\} \\
Deg_{read}(\mathcal{RW}) &= \max_{v \in V} |Read(v)| \\
Rad_{read}(\mathcal{RW}) &= \frac{1}{m} \cdot \max_{u,v \in V} \{dist(u,v) \mid u \in Read(v)\}
\end{aligned}$$

In what follows we use a hierarchy of 2^i-regional matchings in order to design our directories, and show that the complexities of the `Move` and `Find` operations in these directories

depend on the above parameters of the matchings. We then proceed to describe the construction of efficient regional matchings with respect to these parameters (in Sections 6 and 7). The construction has the property that the parameters are independent of m, i.e., the 2^i-regional matchings to be used on levels $1 \leq i \leq \delta$ all have the same radius and degree parameters.

4.2 Regional directories

Our constructions are based on hierarchically organizing the tracking information in *regional directories*. A regional directory is based on defining a "regional address" $\texttt{R_Addr}(\xi)$ for every user ξ. In the hierarchical context, this address represents the most updated *local* knowledge regarding the whereabouts of the user. In particular, the regional address $\texttt{R_Addr}(\xi)$ may be outdated, as ξ may have moved in the meantime to a new location without bothering to update the regional directory. Nevertheless, the structure of the global directory enables one to use the regional address in order to track the current location of the user.

The basic tasks of a regional directory are basically similar to those of a regular (global) directory, namely, to enable the retrieval of the regional address, and to change it whenever needed. For technical reasons, the basic modification tasks are easier to represent in the form of "insert" and "delete" operations, rather than the more natural "move" operation. Thus an *m-regional* directory $\mathcal{RD}$ supports the operations $\texttt{R_find}(\mathcal{RD}, \xi, v)$, $\texttt{R_del}(\mathcal{RD}, \xi, s)$ and $\texttt{R_ins}(\mathcal{RD}, \xi, t)$. These operations are defined as follows.

$\texttt{R_ins}(\mathcal{RD}, \xi, t)$: invoked at the location t, this operation sets t to be the regional address of ξ, i.e., it sets $\texttt{R_Addr}(\xi) \leftarrow t$.

$\texttt{R_del}(\mathcal{RD}, \xi, s)$: invoked at the regional address $s = \texttt{R_Addr}(\xi)$, this operation nullifies the current regional address of ξ, i.e., sets $\texttt{R_Addr}(\xi) \leftarrow nil$.

$\texttt{R_find}(\mathcal{RD}, \xi, v)$: invoked at the vertex v, this operation returns (to node v) the regional address $\texttt{R_Addr}(\xi)$ of the user ξ. This operation is guaranteed to succeed only if $dist(v, \texttt{R_Addr}(\xi)) \leq m$. Otherwise, the operation may *fail*, i.e., it may be that no address is found for ξ. If that happens then an appropriate message is returned to v.

The construction of an m-regional directory is based on an m-regional matching $\mathcal{RW}$. The basic idea is the following. Suppose that the regional address of the user ξ is $s = \texttt{R_Addr}(\xi)$. Then each vertex in the write set $Write(s)$ keeps a pointer $\texttt{Pointer}(\xi)$, pointing to s.

In order to implement the operation $\texttt{R_find}(\mathcal{RD}, \xi, v)$, the searcher v successively queries the vertices in its read set, $Read(v)$, until hitting one that has a pointer $\texttt{Pointer}(\xi)$ leading to the regional address of ξ. In case none of the vertices in $Read(v)$ has the desired pointer, the operation is said to end in failure.

Operation $\texttt{R_del}(\mathcal{RD}, \xi, s)$, invoked at $s = \texttt{R_Addr}(\xi)$, consists of deleting the pointers $\texttt{Pointer}(\xi)$ pointing to s at all the vertices in $Write(s)$. Similarly, operation $\texttt{R_ins}(\mathcal{RD}, \xi, t)$, invoked at the vertex t, consists of inserting pointers $\texttt{Pointer}(\xi)$ pointing to t at all the vertices in $Write(t)$, thus effectively setting $\texttt{R_Addr}(\xi) = t$. The two operations will be performed together, so ξ cannot end up having more than one address in the directory.

A formal presentation of operations R_find, R_del and R_ins is given in Figure 1. The presentation of the algorithms follows the "token language" of [KKM]. That is, the algorithm maintains a "token", which represents the center of activity of the protocol, and moves it around the network. Some algorithms, e.g., Depth-First Search, are very easy to represent in this language.

In the code we use several commands suitable for this language. The first is "**transfer-control-to** v" which means that the token is moved to vertex v. When we mention a variable of the protocol, we refer to the variable at the current location of the token.

When we write "*Local_var* := **remote-read** *Remote_var* at v" while the token is located at u, we mean the following: go from u to v, read variable *Remote_var*, return to u and write the retrieved value into variable *Local_var* at u. At the end of this operation, The center of activity remains at u.

Similarly, "*Remote_var* at v := **remote-write** *Local_var*" means that the value *Local_var* at u is retrieved, and the token carries it from u to v and writes it into *Remote_var* variable at v. The token then returns to u.

It is straightforward to verify the correctness of the above implementation. Hence we have

Lemma 4.2 If $\mathcal{RW}$ is an m-regional matching then the three procedures described above (Figure 1) correctly implement the operations R_find, R_del and R_ins of an m-regional directory. ∎

Lemma 4.3 The implementation of the regional operations in Figure 1 has the following complexities:

(1) $Cost(\texttt{R_find}(\mathcal{RD}, \xi, v)) \leq 2m \cdot Deg_{read}(\mathcal{RW}) \cdot Rad_{read}(\mathcal{RW})$,

(2) $Cost(\texttt{R_del}(\mathcal{RD}, \xi, s)) \leq m \cdot Deg_{write}(\mathcal{RW}) \cdot Rad_{write}(\mathcal{RW})$,

```
R_find(RD, ξ, v):                                        /* invoked at a vertex v */
    repeat forall u ∈ Read(v)
        address ← remote-read Pointer(ξ) from u
    until address ≠ nil or all vertices u are exhausted

R_del(RD, ξ, s):                                         /* invoked at s = R_Addr(ξ) */
    for all u ∈ Write(s)
    remote-write Pointer(ξ) ← nil at u

R_ins(RD, ξ, t)                                          /* invoked at a vertex t */
    for all u ∈ Write(t)
        remote-write Pointer(ξ) ← t at u
```

Figure 1: The three operations of the m-regional directory $\mathcal{RD}$, based on an m-regional matching $\mathcal{RW}$.

(3) $Cost(\texttt{R_ins}(\mathcal{RD}, \xi, t)) \leq m \cdot Deg_{write}(\mathcal{RW}) \cdot Rad_{write}(\mathcal{RW})$.

Proof: To prove (1), observe that each R_find operation is answered after at most $Deg_{read}(\mathcal{RW})$ searches, each involving sending a query and getting a reply along a path of length at most $mRad_{read}(\mathcal{RW})$.

Let us now turn to (2). Note that the operation $\texttt{R_del}(\mathcal{RD}, \xi, s)$ involves deleting pointers to s at all the vertices of $Write(s)$. These deletions require sending an appropriate message from s to all vertices in $Write(s)$. The number of such update messages is at most $Deg_{write}(\mathcal{RW})$, and each of them traverses a distance of at most $mRad_{write}(\mathcal{RW})$. A similar argument applies to (3). ∎

4.3 The dynamic directory

Finally we define our family of *dynamic directories* $\mathcal{D}$ as follows. Let $\delta = \lceil \log Diam(G) \rceil$. For $1 \leq i \leq \delta$, construct a 2^i-regional directory $\mathcal{RD}_i$ based on a 2^i-regional matching as described in the previous subsection. Further, the collection of 2^i-regional matchings used for these regional directories is constructed so that all of them have the same Rad_{read}, Deg_{read}, Rad_{write} and Deg_{write} values, i.e., these parameters are independent of the distance parameter 2^i. (As mentioned earlier, the construction described and analyzed in Sections 6 and 7 enjoys this independence property.)

Each processor v and each user ξ participate in each of the 2^i-regional directories $\mathcal{RD}_i$, for $1 \leq i \leq \delta$. In particular, each vertex v has sets $Write_i(v)$ and $Read_i(v)$ in each $\mathcal{RD}_i$, and each user ξ has a regional address $Addr_i(\xi)$ stored for it in each $\mathcal{RD}_i$.

As discussed earlier, the regional address $v = Addr_i(\xi)$ (stored at the regional directory of level i) does not necessarily reflect the true location of the user ξ, since ξ may have moved in the meantime to a new location v'. (As a rule, the lower the level, the more up-to-date is the regional address.) This potential problem is rectified by maintaining at v a *forwarding trace*, which is a pointer $\texttt{Forward}(\xi) = v'$ pointing at v'. It should be clear that the user may in the meantime have moved further, and is no longer at v'. In fact, the only variable that definitely maintains the true current address of ξ is its lowest level regional address, $Addr_0(\xi)$.

The invariant maintained by the dynamic directory regarding the relationships between the regional addresses stored at the various levels and the pointers is expressed by the following *reachability condition*:

Definition 4.4 The collection $\langle Addr_1(\xi), \ldots, Addr_\delta(\xi)\rangle$ satisfies the *reachability* condition if for every level $1 \leq i \leq \delta$, either $Addr_i(\xi) = Addr(\xi)$ or $Addr_i(\xi)$ stores a pointer $\texttt{Forward}(\xi)$ pointing to the vertex $Addr_{i-1}(\xi)$.

Figure 2 depicts an example in which the reachability condition is maintained.

As users move about in the network, the system attempts to maintain its information as accurately as possible, and avoid having chains of long forwarding traces. This is controlled as follows.

The vertex $s = Addr(\xi)$ hosting the user ξ maintains also the tuple of regional addresses $\langle Addr_1(\xi), \ldots, Addr_\delta(\xi)\rangle$ and a tuple of counters $\langle C_1(\xi), \ldots, C_\delta(\xi)\rangle$. Each counter $C_i(\xi)$ counts the distance traveled by ξ since the last time $addr_i(\xi)$ was updated in $\mathcal{RD}_i$. Another invariant maintained by the updating algorithm is that $0 \leq C_i(\xi) \leq 2^{i-1} - 1$.

A $\texttt{Find}(\mathcal{D}, \xi, v)$ instruction is performed as follows. The querying vertex v successively issues instructions $\texttt{R_find}(\mathcal{RD}_i, \xi, v)$ in the regional directories $\mathcal{RD}_1$, $\mathcal{RD}_2$ etc., until it reaches the first level i on which it succeeds in reaching the regional address $Addr_i(\xi)$. (There must be such a level, since the highest level always succeeds.)

At this point, the searcher v starts tracing the user through the network, starting from $Addr_i(\xi)$, and moving along forwarding pointers. This tracing eventually leads to the real address of the user, $Addr(\xi)$. The procedure $\texttt{Find}(\mathcal{D}, \xi, v)$ is formally described in Figure 3.

A $\texttt{Move}(\mathcal{D}, \xi, s, t)$ operation is carried out as follows. All counters C_i are increased by

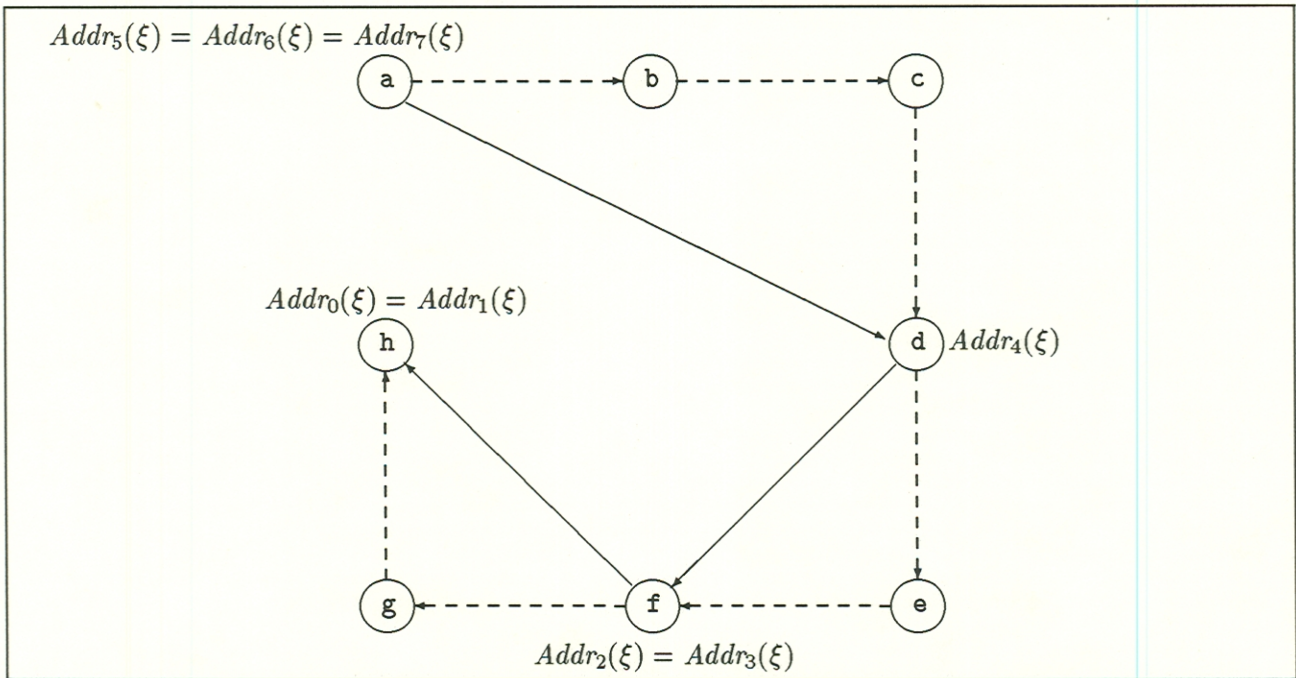

Figure 2: Forwarding pointers and trajectory of a user ξ. The dashed arrows represent the trajectory of the user; the solid arrows represent forwarding pointers.

$i \leftarrow 0$	/* level of the hierarchy */
address $\leftarrow$ *nil*	/* potential pointer to $Addr_i(\xi)$ */
repeat	/* get hold of the $Addr_i(\xi)$ for smallest i */
$i \leftarrow i + 1$	/* increment counter */
address $\leftarrow$ `R_find`$(\mathcal{RD}_i, \xi, v)$	
until *address* $\neq$ *nil*	/* *address* points to $Addr_i(\xi)$ */
transfer-control-to vertex *address*	/* move to $Addr_i(\xi)$ */
repeat	/* trace down ξ along forwarding pointers */
transfer-control-to vertex `Forward`(ξ)	/* trace the user */
until `Forward`$(\xi) = nil$	/* now, located at $Addr(\xi)$ */

Figure 3: Procedure `Find`$(\mathcal{D}, \xi, v)$, invoked at the vertex v.

For $1 \leq i \leq \ell$

  $C_i(\xi) \leftarrow C_i(\xi) + dist(s,t)$

$J \leftarrow \max\{i \mid C_i(\xi) \geq 2^{i-1}\}$

remote-write `Forward`(ξ) $\leftarrow t$ at vertex $Addr_{J+1}(\xi)$ /* pointer to new location */

For $1 \leq i \leq J$ **do:**

  transfer-control-to vertex $Addr_i(\xi)$

  `R_del`($\mathcal{RD}_i, \xi, Addr_i(\xi)$)

end-for

Relocate user ξ to vertex t, along with its $\langle Addr_i \rangle$ and $\langle C_i \rangle$ tuples.

For $1 \leq i \leq J$ **do:**

  `R_ins`($\mathcal{RD}_i, \xi, t$)

  $C_i(\xi) \leftarrow 0$

end-for

Figure 4: Procedure `Move`($\mathcal{D}, \xi, s, t$), invoked at $s = Addr(\xi)$.

$dist(s,t)$. Let C_J be the highest level counter that wraps around 2^{J-1} as a result of this increase. Then we elect to update the regional directories at levels 1 through J. This involves erasing the old listing of ξ in these directories using procedure `R_del` and inserting the appropriate new listing (pointing at t as the new regional address) using `R_ins`. It is also necessary to leave an appropriate forwarding pointer at $Addr_{J+1}(\xi)$ leading to the new location t, and of course perform the actual relocation of the user. The update procedure `Move`($\mathcal{D}, \xi, s, t$) is described in Figure 4.

5 Analysis of the main construction

Lemma 5.1 Procedure `Move` maintains the reachability invariant.

Proof: By inspection on the algorithm, every time that $Addr_i(\xi)$ is changed, the forwarding pointer at $Addr_{i+1}(\xi)$ is updated appropriately. ∎

For every $1 \leq i \leq \delta$ and every user ξ, at any time, the regional address $Addr_i(\xi)$ is either ξ's current address, $Addr(\xi)$, or one of its previous residences. This accounts for the following somewhat stronger claim.

Lemma 5.2 For every $1 \leq i \leq \delta$ and every user ξ, at any time, the path p obtained by starting at the regional address $Addr_i(\xi)$ and tracing the forwarding traces `Forward`(ξ) has the following

properties:

1. p leads to $Addr(\xi)$, and
2. p is no longer than the actual path taken by ξ in its migration from $Addr_i(\xi)$ to $Addr(\xi)$.

Proof: The first property follows directly from the reachability invariant. For the second property, observe that $dist(Addr_i(\xi), Addr_{i-1}(\xi))$, the "span" of the forwarding pointer on the i'th level, can only be smaller than or equal to the actual path taken by ξ in its migration since the last time $Addr_i$ was changed until the last time $Addr_{i-1}$ was changed. ▮

Lemma 5.3 The dynamic directory $\mathcal{D}$ satisfies $Stretch_{find} = O(Deg_{read} \cdot Rad_{read})$.

Proof: Consider a sequence $\bar{\sigma}$ of operations, with a subsequence $\mathcal{F}(\bar{\sigma})$ of `Find` operations. We shall actually prove a stronger claim than the lemma, namely, we will upper bound the worst-case stretch $\frac{Cost(F)}{Opt_cost(F)}$ of any single `Find` operation $F \in \mathcal{F}(\bar{\sigma})$, rather than just the amortized stretch $Stretch_{find}$. Clearly, this worst-case stretch upper-bounds the average stretch, i.e.,

$$Stretch_{find}(\bar{\sigma}) = \frac{\sum_{F\in\mathcal{F}(\bar{\sigma})} Cost(F)}{\sum_{F\in\mathcal{F}(\bar{\sigma})} Opt_cost(F)} \leq \max_{F\in\mathcal{F}(\bar{\sigma})} \frac{Cost(F)}{Opt_cost(F)} .$$

Suppose that a processor v issues an instruction $F = \mathtt{Find}(\mathcal{D}, \xi, v)$ for some user ξ in the network. Recall that $Opt_cost(F) = dist(v, Addr(\xi))$. Let $\beta = \lceil \log dist(v, Addr(\xi)) \rceil$ (i.e., $2^{\beta-1} < dist(v, Addr(\xi)) \leq 2^{\beta}$).

Recall that the querying vertex v successively executes find operations $RF_i = \mathtt{R_find}(\mathcal{RD}_i, \xi, v)$ for $i = 1, 2, \ldots$, until it reaches the first level i on which it succeeds in reaching $Addr_i(\xi)$. Success on level i is guaranteed if $dist(v, Addr_i(\xi)) \leq 2^i$. Since $dist(Addr(\xi), Addr_{\beta+1}(\xi)) \leq 2^{\beta}$, by the triangle inequality we deduce that

$$dist(v, Addr_{\beta+1}(\xi)) \leq dist(v, Addr(\xi)) + dist(Addr(\xi), Addr_{\beta+1}(\xi)) \leq 2^{\beta} + 2^{\beta} = 2^{\beta+1}.$$

It follows from the definition of a regional directory that the operation $RF_{\beta+1} = \mathtt{R_find}(\mathcal{RD}_{\beta+1}, \xi, v)$ must succeed, hence the first level i on which RF_i succeeds satisfies $i \leq \beta + 1$,

At this point v starts tracing the user through the network, starting from $Addr_i(\xi)$, and moving along forwarding pointers. By Lemma 5.2 this tracing leads to $Addr(\xi)$ along a path no longer than $C_i(\xi)$, which in turn is kept no larger than 2^{i-1}.

Therefore the actual communication cost $Cost(F)$ for the search operation $F = \mathtt{Find}(\mathcal{D}, \xi, v)$, (which equals the total length of the combined path traversed by the search messages), sat-

isfies by Lemma 4.3

$$\begin{aligned} Cost(F) &\leq 2 \cdot 2^\beta + \sum_{i=1}^{\beta+1} Cost(RF_i) \leq 2 \cdot 2^\beta + \sum_{i=1}^{\beta+1} (2 \cdot 2^i \cdot Deg_{read} \cdot Rad_{read}) \\ &\leq (1/2 + 2 \cdot Deg_{read} \cdot Rad_{read}) \cdot 2^{\beta+2} \\ &\leq 8(1/2 + 2 \cdot Deg_{read} \cdot Rad_{read}) \cdot dist(v, Addr(\xi)), \end{aligned}$$

establishing

$$\frac{Cost(F)}{Opt_cost(F)} \leq 8(1/2 + 2 \cdot Deg_{read} \cdot Rad_{read}).$$

∎

Lemma 5.4 The dynamic directory $\mathcal{D}$ satisfies
$Stretch_{move} = O(Rad_{write} \cdot Deg_{write} \cdot \log Diam(G))$.

Proof: Consider any sequence $\bar{\sigma}$ of operations, and its subsequence $\mathcal{M}(\bar{\sigma}) = (M_1, \ldots, M_k)$ of `Move` operations, where $M_i = (\mathcal{D}, \xi, s^i, s^{i+1})$. Let $\rho_{\mathcal{M}}$ denote the total length of the path taken by ξ since its insertion into the system,

$$\rho_{\mathcal{M}} = \sum_{1 \leq i \leq k} dist(s^i, s^{i+1}).$$

Observe that

$$Opt_cost(\mathcal{M}(\bar{\sigma})) = \sum_{1 \leq i \leq k} Reloc(\xi, s^i, s^{i+1}) \geq \rho_{\mathcal{M}}.$$

Let us consider one move operation $M = (\mathcal{D}, \xi, s, t)$. Let $J(M)$ denote the highest counter that wrapped around in this move. Denote $s_i = Addr_i(\xi)$ prior to that move, for every $1 \leq i \leq J+1$. In this update, $Addr_i(\xi)$ is changed from s_i to t, for every $1 \leq i \leq J$. We will charge the entire cost of the operation M to the index $J(M)$.

Let us now analyze this cost. The first thing we do is sending a message to vertex s_i and performing `R_del`$(\mathcal{RD}_i, \xi, s_i)$ from s_i, for all $i \leq J$. Observe that for every i, $dist(s, s_i) \leq 2^{i-1}$, which bounds the cost of these messages. By Lemma 4.3, the overall communication complexity of `R_del`$(\mathcal{RD}_i, \xi, s_i)$ is at most $2^i \cdot Rad_{write} \cdot Deg_{write}$. Thus overall we pay at most

$$\sum_{1 \leq i \leq J} (2^{i-1} + 2^i \cdot Rad_{write} \cdot Deg_{write}) \leq 2^{J+1}(1/2 + Rad_{write} \cdot Deg_{write}).$$

Sending a message to vertex s_{J+1} (informing it to change the value of the pointer `Forward`(ξ) so that it points to t and not to s_i) costs at most 2^J by the same argument. Relocating the user ξ to vertex t costs us $Reloc(\xi, s, t)$. Finally, performing `R_ins`$(\mathcal{RD}_i, \xi, t)$ from t, for all $i \leq J$, costs at most $2^{J+1} \cdot Rad_{write} \cdot Deg_{write}$ by Lemma 4.3 again.

Overall, we charge the index $J(M)$ a total of $Reloc(\xi, s, t) + 2^{J+2} \cdot (1/2 + Rad_{write} \cdot Deg_{write})$.

However, notice that whenever an index J is charged for a move in the sequence $\mathcal{M}(\bar{\sigma})$, the counter C_J has completed at least one entire undisturbed wrap. Thus J could have been selected as the index $J(M)$ of an operation in the sequence (and thus charged for the operation) at most $\frac{\rho_{\mathcal{M}}}{2^{J-1}}$ times.

Summing over all updates of all regional directories, we get the following bounds.

$$
\begin{aligned}
Cost(\mathcal{M}(\bar{\sigma})) &\leq \sum_{i=1}^{k} \left(Reloc(\xi, s^i, s^{i+1}) + 2^{J(M_i)+2} \cdot (1/2 + Rad_{write} \cdot Deg_{write}) \right) \\
&\leq Opt_cost(\mathcal{M}(\bar{\sigma})) + \sum_{J=1}^{\delta} \frac{\rho_{\mathcal{M}}}{2^{J-1}} \cdot 2^{J+2} \cdot (1/2 + Rad_{write} \cdot Deg_{write}) \\
&= Opt_cost(\mathcal{M}(\bar{\sigma})) + 8 \cdot \delta \cdot \rho_{\mathcal{M}} \cdot (1/2 + Rad_{write} \cdot Deg_{write}).
\end{aligned}
$$

Hence

$$
Stretch_{move}(\bar{\sigma}) = \frac{Cost(\mathcal{M}(\bar{\sigma}))}{Opt_cost(\mathcal{M}(\bar{\sigma}))} \leq 1 + 8\lceil \log Diam(G) \rceil (1/2 + Rad_{write} \cdot Deg_{write}).
$$

▮

Lemma 5.5 The directory constructed as above can be implemented for N users using a total of $O((N \cdot Deg_{write} + n \cdot Deg_{read}) \log Diam(G) \cdot \log n)$ memory bits throughout the network.

Proof: The regional directory implementation involves the following space requirements. Each user ξ posts its address $v = Addr(\xi)$ at the vertices of $Write(v)$. Summing over N users and δ levels, this gives a total memory of $O(N \cdot Deg_{write} \cdot \log Diam(G) \cdot \log n)$. In addition, each processor v needs to know the identity of the vertices in $Read(v)$. This requires a total additional amount of $O(Deg_{read} \cdot n \cdot \log Diam(G) \cdot \log n)$ bits. ▮

Summarizing the above three lemmas, we get

Lemma 5.6 Given an appropriate family of regional matching, the directory $\mathcal{D}$ constructed as above satisfies $Stretch_{find} = O(Deg_{read} \cdot Rad_{read})$ and $Stretch_{move} = O(Rad_{write} \cdot Deg_{write} \cdot \log n)$, and requires a total of $O((N \cdot Deg_{write} + n \cdot Deg_{read}) \log^2 n)$ memory bits throughout the network in order to handle N users. ▮

The next section develops the family of regional matchings $\mathcal{RW}_{m,k}$ that are actually used for the regional directories. Essentially, we prove the following theorem.

Theorem 5.7 For all $m, k \geq 1$, it is possible to construct an m-regional matching $\mathcal{RW}_{m,k}$ with

$$
Deg_{read}(\mathcal{RW}_{m,k}) \leq k \qquad (1)
$$

$$Rad_{read}(\mathcal{RW}_{m,k}) \leq 2k+1 \quad (2)$$
$$Deg_{write}(\mathcal{RW}_{m,k}) \leq n^{2/k} \quad (3)$$
$$Rad_{write}(\mathcal{RW}_{m,k}) \leq 2k+1 \quad (4)$$

▮

Using Lemma 5.6 and the last theorem we get

Theorem 5.8 Using a family of regional matchings $\mathcal{RW}_{m,k}$ as in Lemma 5.6, the directory $\mathcal{D}$ constructed as above satisfies $Stretch_{find} = O(k^2)$ and $Stretch_{move} = O(k \cdot n^{2/k} \cdot \log n)$, and requires a total of $O((N \cdot n^{2/k} + n \cdot k) \log^2 n)$ memory bits throughout the network in order to handle N users. ▮

Now, consider the directory $\mathcal{D}$ obtained by picking $k = \log n$.

Corollary 5.9 The directory $\mathcal{D}$ satisfies $Stretch_{find} = O(\log^2 n)$ and $Stretch_{move} = O(\log^2 n)$ and uses a total of $O(N \cdot \log^2 n + n \cdot \log^3 n)$ memory bits for handling N users. ▮

6 Constructing a regional matching

In this section we describe the construction of an efficient m-regional matching. In the next section we analyze the radius and degree parameters of the matching and thus prove Theorem 5.7.

6.1 Preliminaries

The *j-neighborhood* of a vertex $v \in V$ is defined as $\Gamma_j(v) = \{w \mid dist(w,v) \leq j\}$.

For a vertex $v \in V$, let $Rad(v,G) = \max_{w \in V}(dist_G(v,w))$. Let Rad_G denote the *radius* of the network, i.e., $\min_{v \in V}(Rad(v,G))$. A *center* of G is any vertex v realizing the radius of G (i.e., such that $Rad(v,G) = Rad(G)$). In order to simplify some of the following definitions we avoid problems arising from 0-diameter or 0-radius graphs, by defining $Rad(G) = Diam(G) = 1$ for the single-vertex graph $G = (\{v\}, \emptyset)$. Observe that for every graph G, $Rad(G) \leq Diam(G) \leq 2Rad(G)$. (In all of the above notations we may sometimes omit the reference to G where no confusion arises.)

Let us now introduce some definitions relevant for covers. Given a set of vertices $S \subseteq V$, let $G(S)$ denote the subgraph induced by S in G. A *cluster* is a subset of vertices $S \subseteq V$ such that $G(S)$ is connected. A *cover* is a collection of clusters $\mathcal{S} = \{S_1, \ldots, S_m\}$ such that $\bigcup_i S_i =$

```
∀v ∈ V, Write(v) ← ∅, Read(v) ← ∅
S ← {Γ_m(v) | v ∈ V}                 /* S is the basic cover containing all m-neighborhoods */
R ← V                                 /* R is the set of "remaining" vertices */
repeat
   Call Procedure Cover(R,S,T)        /* the output T is a cover of R */
   Call Procedure Update(R,S,T)       /* low-degree vertices are deleted from R */
until R = ∅
```

Figure 5: Algorithm `Main`.

V. We use $Rad(v, \mathcal{S})$ (respectively, $Rad(\mathcal{S})$, $Diam(\mathcal{S})$) as a shorthand for $Rad(v, G(\mathcal{S}))$ (resp., $Rad(G(\mathcal{S}))$, $Diam(G(\mathcal{S}))$). Given a cover $\mathcal{S}$, let $Diam(\mathcal{S}) = \max_i Diam(S_i)$ and $Rad(\mathcal{S}) = \max_i Rad(S_i)$. For every vertex $v \in V$, let $deg_{\mathcal{S}}(v)$ denote the degree of v in the hypergraph $(V, \mathcal{S})$, i.e., the number of occurrences of v in clusters $S \in \mathcal{S}$.

Given two covers $\mathcal{S} = \{S_1, \ldots, S_m\}$ and $\mathcal{T} = \{T_1, \ldots, T_k\}$, we say that $\mathcal{T}$ *subsumes* $\mathcal{S}$ if for every $S_i \in \mathcal{S}$ there exists a $T_j \in \mathcal{T}$ such that $S_i \subseteq T_j$.

6.2 The construction of a regional matching

In this subsection we present the algorithm `Main`, whose task is to construct an m-regional matching $\mathcal{RW}_{m,k}$. The algorithm receives as input a graph $G = (V, E)$, $|V| = n$, and an integer k. The algorithm starts with constructing the *basic cover* $\mathcal{S} = \{\Gamma_m(v) \mid v \in V\}$. This cover satisfies $Rad(\mathcal{S}) \leq m$. The algorithm maintains the set of "remaining" vertices R. Initially, $R = V$. The algorithm terminates once $R = \emptyset$. The algorithm operates in at most k phases. Each phase consists of the activation of two procedures, `Cover` and `Update`. Algorithm `Main` is formally described in Figure 5.

Procedure `Cover` constructs the covers that are used as a basis for the regional matching. At phase i, the procedure constructs a cover $\mathcal{T}$, which subsumes the basic cover $\mathcal{S}$. For each set $T \in \mathcal{T}$, it also specifies a *kernel* $K(T) \subseteq T$ and a *center* $\ell(T) \in K(T)$, with some special properties, as summarized in Lemma 7.7.

The procedure is a variant of a similar algorithm presented in [P1, P2]. It operates in iterations during which it picks a cluster in the original cover $\mathcal{S}$ and starts merging it with intersecting clusters until reaching a certain sparsity condition. These iterations proceed until $\mathcal{S}$ is exhausted. The result is a cover $\mathcal{T}$ subsuming $\mathcal{S}$. (The original cover $\mathcal{S}$ is restored for the next phase.) Procedure `Cover` is formally described in Figure 6.

$\tilde{\mathcal{S}} \leftarrow \mathcal{S}$
$\mathcal{T} \leftarrow \emptyset$
repeat
  Select an arbitrary cluster $T \in \tilde{\mathcal{S}}$.
  $\tilde{\mathcal{S}} \leftarrow \tilde{\mathcal{S}} - \{T\}$
  repeat
    $K \leftarrow T$
    $Q \leftarrow \{S' \mid S' \in \tilde{\mathcal{S}},\ S' \cap K \neq \emptyset\}$.
    $T \leftarrow K \cup \bigcup_{S' \in Q} S'$
    $\tilde{\mathcal{S}} \leftarrow \tilde{\mathcal{S}} - Q$
  until $|R \cap T| \leq n^{1/k} |R \cap K|$
  Set $\mathcal{T} \leftarrow \mathcal{T} \cup \{T\}$
  $K(T) \leftarrow K$
  set $\ell(T)$ to be any center of $G(K(T))$.
until $\mathcal{S} = \emptyset$
Output the collection of sets $\mathcal{T}$

Figure 6: Procedure $\mathtt{Cover}(R, \mathcal{S}, \mathcal{T})$, constructing a cover $\mathcal{T}$ of R.

Procedure `Update` is responsible for the construction of the m-regional matching $\mathcal{RW}_{m,k}$ from the covers produced by Procedure `Cover`. Thus this procedure has to construct the sets *Read*(v) and *Write*(v) for every vertex v. At phase i, the procedure first identifies the vertices $u \in R$ with small degrees, namely, such that $deg_{\mathcal{T}}(u) \leq n^{2/k}$. For such vertices, the set *Write*(u) is now determined to be the collection of centers $\ell(T)$ of all clusters $T \in \mathcal{T}$ such that $u \in T$. The procedure next handles the *Read* sets. The cover $\mathcal{T}$ produced by Procedure `Cover` at each iteration i subsumes the original cover $\mathcal{S}$, that is, for every vertex $v \in V$ there is a cluster $T \in \mathcal{T}$ such that $\Gamma_m(v) \subseteq T$. Consequently, the procedure adds the center of T, $\ell(T)$, to the set *Read*(v). (In case there are several appropriate clusters, the procedure selects one arbitrarily.) Thus the set *Write*(v) is determined once and for all at some phase i, while the set *Read*(v) is constructed gradually, adding one vertex at each phase. Procedure `Update` is formally described in Figure 7.

$\forall v \in V$,

  pick some $T \in \mathcal{T}$ such that $\Gamma_m(v) \subseteq T$

  $Read(v) \leftarrow Read(v) \bigcup \{\ell(T)\}$,

$\forall v \in R$,

  if $deg_{\mathcal{T}}(v) \leq n^{2/k}$ **then**

    $Write(v) = \{\ell(T) \mid T \in \mathcal{T},\ v \in T\}$.

    $R \leftarrow R - \{v\}$.

Figure 7: Procedure `Update`.

7 Correctness and analysis of the regional matching

7.1 Properties of Procedure `Cover`

The properties of Procedure `Cover` are summarized by the following lemma.

Lemma 7.1 Given a graph $G = (V, E)$, $|V| = n$, an integer k, a subset $R \subseteq V$ and any cover $\mathcal{S}$, the collection $\mathcal{T}$ constructed by Procedure `Cover`$(R, \mathcal{S}, \mathcal{T})$ is a cover of R and it satisfies the following properties:

(1) $\mathcal{T}$ subsumes $\mathcal{S}$,

(2) $\sum_{v \in R} deg_{\mathcal{T}}(v) \leq n^{1/k}|R|$, and

(3) $Rad(\ell(T), T) \leq (2k+1)Rad(\mathcal{S})$ for every $T \in \mathcal{T}$.

Proof: The proof mostly parallels that of the partitioning algorithm GV of [P2]. The fact that the input $\mathcal{S}$ is a cover implies in particular that every $S \in \mathcal{S}$ is a cluster (i.e., its induced graph $G(S)$ is connected) and that $\bigcup \mathcal{S} = V$. Consequently, the construction process guarantees that every set T added to $\mathcal{T}$ is a cluster. Furthermore, every cluster $S \in \mathcal{S}$ is merged into some cluster of $\mathcal{T}$. This implies that the clusters of $\mathcal{T}$ contain all vertices of V, so $\mathcal{T}$ is a cover, and moreover, it subsumes $\mathcal{S}$, and Property (1) holds.

Property (2) is derived as follows. From the termination condition of the internal loop it is immediate that $|R \cap T| \leq n^{1/k}|R \cap K(T)|$ for every $T \in \mathcal{T}$. Therefore

$$\sum_{v \in R} deg_{\mathcal{T}}(v) = \sum_{T \in \mathcal{T}} |R \cap T| \leq \sum_{T \in \mathcal{T}} n^{1/k}|R \cap K(T)|.$$

We now argue the following.

Claim 7.2 $K(T) \cap K(T') = \emptyset$ for every $T, T' \in \mathcal{T}$.

Proof: Suppose, seeking to establish a contradiction, that there is a vertex v such that $v \in K(T_i) \cap K(T_j)$. Without loss of generality suppose that T_i was created before T_j. Since $v \in K(T_j)$, there must be a cluster S' such that $v \in S'$ and S' was still in $\tilde{\mathcal{S}}$ when the algorithm started constructing T_j. But every such cluster S' satisfies $S' \cap K(T_i) \neq \emptyset$, and therefore the final construction step creating T_i from $K(T_i)$ should have merged S' into T_i and eliminated it from $\mathcal{S}$; a contradiction. ∎

As a result of the last claim we get

$$\sum_{v \in R} deg_{\mathcal{T}}(v) \leq n^{1/k} \left| R \cap \bigcup_{T \in \mathcal{T}} K(T) \right| \leq n^{1/k} |R|.$$

Finally we analyze the increase in the radius of clusters in the cover. Consider some iteration of the main loop of Procedure `Cover` in Figure 6, starting with the selection of some cluster $T \in \tilde{\mathcal{S}}$. Let J denote the number of times the internal loop was executed. Denote the initial set T by T_0. Denote the set T (respectively, K) constructed on the i'th internal iteration ($1 \leq i \leq J$) by T_i (resp., K_i). Note that for $1 \leq i \leq J$, T_i is constructed on the basis of K_i, and $K_i = T_{i-1}$. We proceed along the following chain of claims.

Claim 7.3 $|R \cap T_i| \geq n^{i/k}$ for every $0 \leq i \leq J-1$, and strict inequality holds for $i \geq 1$.

Proof: By induction on i. The claim is immediate for $i = 0$. Assuming the claim for $i - 1 \geq 0$, it remains to prove that

$$|R \cap T_i| > n^{1/k} |R \cap T_{i-1}|,$$

which follows directly from the fact that the termination condition of the internal loop was not met. ∎

Corollary 7.4 $J \leq k$.

Claim 7.5 For every $0 \leq i \leq J$, $Rad(T_i) \leq (2i+1)Rad(\mathcal{S})$.

Proof: We first note that for $1 \leq i \leq J$,

$$Rad(T_i) \leq Rad(K_i) + Diam(\mathcal{S}) \leq Rad(K_i) + 2Rad(\mathcal{S}),$$

since T_i is created from K_i by merging into it some neighboring clusters from $\mathcal{S}$. The proof now follows by straightforward induction on i, since $T_0 = T \in \mathcal{S}$ and $K_i = T_{i-1}$ for $1 \leq i \leq J$. ∎

Since $K(T) = K(T_J) = T_{J-1}$, it follows from Corollary 7.4 and Claim 7.5 that

Corollary 7.6 For every $T \in \mathcal{T}$, $Rad(K(T)) \leq (2k-1)Rad(\mathcal{S})$.

This finally enables us to prove the last property of the Lemma, upon noting that $Rad(\ell(T), T) \leq Rad(K(T)) + 2Rad(\mathcal{S})$.

This completes the proof of Lemma 7.1. ∎

7.2 Properties of Algorithm `Main`

The properties of the algorithm are summarized by the following lemma. Let $\mathcal{T}^i$ be the cover $\mathcal{T}$ produced at phase i. In addition, let R^i denote the contents of the set R at the beginning of phase i, and let $V^i = R^i \setminus R^{i+1}$, i.e., the set of vertices assigned a read set and deleted from R by Procedure `Update` in phase i.

Lemma 7.7 Given a graph $G = (V, E)$, $|V| = n$ and an integer k, Algorithm `Main` performs at most k phases, and the covers $\mathcal{T}^i$ and sets V^i constructed in these phases satisfy the following properties, for every i:

(1) $\mathcal{T}_i$ subsumes the basic cover $\mathcal{S} = \{\Gamma_m(v) \mid v \in V\}$,

(2) $deg_{\mathcal{T}^i}(v) \leq n^{2/k}$ for every $v \in V^i$,

(3) $deg_{\mathcal{T}^i}(v) > n^{2/k}$ for every $v \in R^{i+1}$, and

(4) $Rad(\ell(T), T) \leq (2k+1)m$ for every $T \in \mathcal{T}^i$.

Proof: Property (1) is immediate from Property (1) of Lemma 7.1. Properties (2) and (3) follow directly from the rule by which Procedure `Update` determines the vertices of V^i in each phase i. Property (4) follows from Property (3) of Lemma 7.1 and the fact that the basic cover $\mathcal{S}$ constructed by the algorithm satisfies $Rad(\mathcal{S}) \leq m$.

It remains to bound the number of phases performed by the algorithm. This bound relies on the following observations. By Property (2) of Lemma 7.1 and Property (3) of the current lemma, in every phase i, at most $\frac{|R^i|}{n^{1/k}}$ vertices of R^i remain in the set R^{i+1}, i.e., are not moved to the set V^i. Consequently $|R^i| \leq n^{1-i/k}$. Hence R is exhausted after no more than k phases of Algorithm `Main`. ∎

7.3 Properties of Procedure `Update`

First let us comment that procedure `Update` never gets "stuck". The only potentially problematic step is the first, namely, picking a cluster $T \in \mathcal{T}$ such that $\Gamma_m(v) \subseteq T$. Such a

cluster exists by Property (1) of Lemma 7.7.

The correctness of the constructed regional matching $\mathcal{RW}_{m,k}$ is proved by the following lemma, relying on the properties of the cover $\mathcal{T}^i$ (as stated in Lemma 7.7).

Lemma 7.8 For any two vertices u, v such that $dist(u, v) \leq m$, $Read(u) \bigcap Write(v) \neq \emptyset$.

Proof: Suppose that $dist(v, u) \leq m$ for some processors v and u. Hence $u \in \Gamma_m(v)$. By Lemma 7.7 there is some $1 \leq j \leq k$ such that $u \in V^j$. Let $T \in \mathcal{T}^j$ be the cluster picked for $Read(v)$ during the first part of Procedure `Update` in phase j. Then the procedure adds $\ell(T)$ to $Read(v)$. On the other hand note that T is selected such that $\Gamma_m(v) \subseteq T$, and thus $u \in T$. Since $T \in \mathcal{T}^j$, $\ell(T)$ is added during the second part of Procedure `Update` to $Write(u)$ as well. Thus, $\ell(T) \in Read(u) \bigcap Write(v)$. ▮

This shows that the structure $\mathcal{RW}_{m,k}$ described above is indeed an m-regional matching. It remains to prove that this structure meets the requirements of Thm. 5.7.

Proof of Theorem 5.7: Property (1) follows from the fact that Algorithm `Main` performs at most k phases (by Lemma 7.7), and each of these phases adds exactly one vertex to $Read(v)$.

Property (3) follows from the construction rule of $Write(v)$ in Procedure `Update` and from Property (2) of Lemma 7.7.

Finally, to prove Properties (2) and (4), note that any vertex added to the sets $Read(v)$ or $Write(v)$ is the center $\ell(T)$ of some cluster T such that $v \in T$, and therefore the distance from v to such vertex is no larger than $Rad(\ell(T), T)$. The claims now follow from Property (4) of Lemma 7.7. ▮

8 Discussion

The setting considered in this paper does not allow the operations of inserting a new user into the system or deleting an existing user from the system. This is because these operations require considerably heavy updates on the directories, which cannot be charged directly to any "inherent cost." It is possible to include such operations in the proposed framework by making the assumption that inserting a new user is as expensive as moving the user across the entire network, i.e., its inherent cost is at least $Diam(G)$. A similar assumption has to be made for deletions of users from the system.

As mentioned earlier, we made the assumption that the `Find` and `Move` operations are

performed "sequentially," i.e., that there is enough time for the system to complete its operation on one request before getting the next one. Interesting problems arise when many operations are issued simultaneously. Specifically, problems may occur if someone attempts to contact a user *while* it is moving. It is necessary to ensure that the searcher will eventually be able to reach the moving user, even if that user moves repeatedly. This calls for applying elaborate locking and sweeping techniques, somewhat resembling ideas from atomicity theory and concurrency control. Generalizing the present solution to the concurrent case will be the subject of subsequent work.

An interesting area of research concerns developing good policies for moving servers in the network, in response to a stream of requests, in order to achieve best performance. In the literature on sequential algorithms this problem is known as the "k-server" problem (cf. [MMS]). In the distributed setting there are additional complicating factors, having to do with the effects of locality and partial information. Even though this question is not dealt with in this paper, our contribution is in showing that the limitation of partial information is not very restrictive, in the sense that one can get by paying no more than a polylogarithmic factor in performance compared to an algorithm which obtains full information for free.

Acknowledgments

The authors thank Michael Fischer for the stimulating discussion that triggered this research, and Richard Karp for his helpful comments.

References

[KKM] E. Korach, S. Kutten and S. Moran, A Modular Technique for the Design of Efficient Distributed Leader Finding Algorithms, *ACM Trans. on Prog. Lang. and Syst.*, to appear. (Preliminary version appeared in *4th ACM Symp. on Principles of Distributed Computing*, Minaki, Ontario, Canada, 1985, pp. 163-174.)

[KV] E. Kranakis and P.M.B. Vitányi, A Note on Weighted Distributed Match-Making, *AWOC* 1988, LNCS 319, pp. 361–368.

[LEH] K.A. Lantz, J.L. Edighoffer and B.L. Histon, Towards a Universal Directory Service, *4th ACM Symp. on Principles of Distributed Computing*, 1985, pp. 261–271.

[MMS] M.S. Manasse, L.A. McGeoch and D.D. Sleator, Competitive Algorithms or Online Problems, *20th ACM Symp. on Theory of Computing*, May 1988.

[MV] S.J. Mullender and P.M.B. Vitányi, Distributed Match-Making, *Algorithmica* **3**, (1988), pp. 367–391.

[P1] D. Peleg, Sparse Graph Partitions, Report CS89-01, Dept. of Applied Math., The Weizmann Institute, Rehovot, Israel, February 1989.

[P2] D. Peleg, Distance-Dependent Distributed Directories, Report CS89-10, Dept. of Applied Math., The Weizmann Institute, Rehovot, Israel, May 1989.